IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Applicant:	Harold E. Helson
Serial No.:	To Be Assigned
Filed:	Herewith (This application claims the benefit of U.S. Provisional Application Serial No. 60/119,654 entitled STRUCTURE DIAGRAM GENERATION, filed on February 11, 1999.)
Title:	ENHANCING STRUCTURE DIAGRAM GENERATION

Box Patent Application
Assistant Commissioner for Patents
Washington, DC 20231

09502133.021100

COVER SHEET FOR SOURCE CODE APPENDIX

Dear Sir:

Enclosed for filing in the above-referenced patent application is the following document:

1. Source Code Appendix, 30 pages.

The following is the inventor's residence:
69 Bartlett Avenue, Arlington, MA 02476.

Respectfully submitted,

Dated: February 11, 2000

Jason A. Reyes

Jason A. Reyes
Registration No. 41,513
Attorney for Applicant

Hale and Dorr LLP
60 State Street
Boston, MA 02109
Tel.: (617) 526-6010
Fax: (617) 526-5000

Attorney Docket No. 103544.127

EXPRESS MAIL LABEL NO. EM259723534US
DATE OF DEPOSIT February 11, 2000

```
/*
     File:          :sdg:sdg_ringDesign.cpp

     Contains:      Computes coordinates of ring systems given a ring strategy.

     Copyright:      © 1996-2000 CambridgeSoft Corp., all rights reserved.

     $Header: /ChemDraw/Src/sdg/sdg_ringDesign.cpp 41    12/23/99 6:32p Jsb $
/*
+---------------------------------------------------------------------------------------+
HEH  01/20/99  CDBR-6450: Changed m_asRingAtomsPlaced_frg into m_asRingAtomsPlaced_RDU.      |
HEH  01/15/99  RD_AttachPeeledBridge(): When choosing bridge position, penalize linear bds.|
HEH  01/14/99  RD_AttachPeeledBridge(): Include bds adjacent to border ats in congest.calc.|
HEH  01/07/99  ComputeCongestion(): Replace ad hoc in-place code with calling Red_Potent().|
HEH  01/05/99  Lengthen or contract bridge to avoid overlap with already-laid down parts.   |
HEH  12/21/98  CFBR-4853: RD_AttachPeeledBridge(): Draw bridge on less congested side.      |
HEH  12/13/98  Added DYNAMIC ring strategy.
|
HEH  09/02/97  Added RA_AreAtomsOrBondsContiguousAboutRing(). Moved RingTransit to CC.      |
HEH  07/29/96  RD_DesignRing(): Clear CFS_definedV of spiro atoms at end of RDU.
|
HEH  07/29/96  RD_MakeSimpleCore(): Add RINGS_REST_ON_FLAT_EDGE
|
HEH  07/19/96  New class RingTransit:: supplants TraverseRing().  ChasePolygon() becomes     |
                    obsolete.
|
HEH  07/19/96  Ring drawing order was determined in ring design; now in ring strategy.       |
HEH  07/19/96  New fn RD_MakeSimpleCore(); MakeSimpleRingSystem() is obsolete.
|
HEH  07/19/96   Handles bridges.  New fn RD_AttachPeeledBridge().
|
HEH  07/19/96   Renamed AttachRing() to RD_AttachSimpleRing().
|
HEH  07/19/96   New fn RD_AttachThing() places a ring's atoms and calculates CFS's given        |
                    a vector of coordinates.  Extracted from old AttachRing() so as to
treat  |
                    the commonality between peeled simple and peeled bridge.
|
+------------------------------------------------------------------------------------+
*/


//-----------------------------------------------------------------------------------
class RD_BridgeCongestionEnvironment
{
public:
                              RD_BridgeCongestionEnvironment (SDG &c, int
ringNum, ccRingTransit &ringTransit, ATOMNO aOuter_CW, ATOMNO aOuter_CCW, SREF asUndrawnAtoms, SREF
asDrawnAtoms, sdgFloat bdLen);
    sdgFloat                ComputeCongestion();
    bool                   IsCongested()   const   { return m_congestion > 20.; }

    int                     m_ringNum;
    ccRingTransit&         m_ringTransit;
    ccPoint2D               m_P1, m_P2;
    vector<ccPoint2D>      m_coords;
    int                     m_numAtsToDraw;
    sdgFloat               m_bdLen;
    const ccSet&          m_asUndrawnAtoms,
                               m_asDrawnAtoms;
    ATOMNO                      m_aOuter_CW, m_aOuter_CCW;
    sdgFloat                m_congestion;   // squirreled copy of value found in
ComputeCongestion().
    vector<ccPoint2D>     m_trialCoords;
    sdgFloat                m_polyPhi;
    SDG&                     C;
};
//--------------------------------------------------------------------------------------------
RD_BridgeCongestionEnvironment::RD_BridgeCongestionEnvironment (SDG &c, int ringNum, ccRingTransit
&ringTransit, ATOMNO aOuter_CW, ATOMNO aOuter_CCW, SREF asUndrawnAtoms, SREF asDrawnAtoms, sdgFloat
bdLen)
    :    C                              (c)
    ,    m_ringNum                     (ringNum)
    ,    m_ringTransit            (ringTransit)
    ,    m_numAtsToDraw          (asUndrawnAtoms.NMems() + 2)    // includes the two border
atoms
    ,    m_coords                      (asUndrawnAtoms.NMems() + 2)    // ditto
    ,    m_bdLen                       (bdLen)
    ,    m_asUndrawnAtoms        (asUndrawnAtoms)
    ,    m_asDrawnAtoms          (asDrawnAtoms)
    ,    m_aOuter_CW                   (aOuter_CW)
    ,    m_aOuter_CCW            (aOuter_CCW)
    ,    m_trialCoords           ((asUndrawnAtoms | asDrawnAtoms).Last() + 1)
{
    ASSERT (m_numAtsToDraw == m_asUndrawnAtoms.NMems() + 2);        // m_numAtsToDraw includes
the two drawn rooted atoms
    m_P1 = C.GetVXY (m_aOuter_CW);
```

```
	m_P2 = C.GetVXY (m_aOuter_CCW);
	const bool	LVal = C.RD_OpenPolygon (m_P1, m_P2, m_numAtsToDraw, m_bdLen,
kccCounterClockwise, m_coords, &m_polyPhi);

	m_ringTransit.MoveTo (m_aOuter_CCW);
	for (int x = 2;  x < m_numAtsToDraw;  x++)	// skip the two rooted atoms
	{
		m_ringTransit.Advance();
		m_trialCoords [m_ringTransit.Curr()] = m_coords [x];
		ASSERT (m_asUndrawnAtoms.IsMem (m_ringTransit.Curr()));
	}
	ATOMNO  a;
	LOOP_SET (m_asDrawnAtoms, a)	// avoid executing this loop many times by moving
m_trialCoords outside of this object
		m_trialCoords [a] = C.GetVXY (a);
}
//-----------------------------------------------------------------------------------------
sdgFloat RD_BridgeCongestionEnvironment::ComputeCongestion()
{
	const ccSet	asTwoBorderAtoms = ccMakeSet (m_aOuter_CW, m_aOuter_CCW);
	const ccSet	asInterestingDrawnAtoms	= m_asDrawnAtoms - asTwoBorderAtoms;
	const ccSet bsInterestingDrawnBonds	= C.CT.JoiningBonds (m_asDrawnAtoms),
				bsInterestingUndrawnBonds = C.CT.JoiningBonds (m_asUndrawnAtoms |
asTwoBorderAtoms);

	m_congestion = C.Potential_BB ( bsInterestingUndrawnBonds,	m_trialCoords,
						bsInterestingDrawnBonds,
m_trialCoords, 3.0);
	CDBG2 ( sdgOut ("ComputeCongestion: total congestion = %8.3lf\n") << m_congestion; )
```

```
    return m_congestion;
}
//-------------------------------------------------------------------------------------


/*
+===================================================================================+
|
|
| LinearAnglePenalty    Calculate a penalty for near-linear bonds.
|
|
|
+-------------------------------------------------------------------------------------+
|  Penned by H.Helson, 1/15/99.
|
+===================================================================================+
*/
static int  LinearAnglePenalty (const sdgFloat &ang, int threshhold_deg = 120)
{
    const int      interiorBondAngle_deg = 180 - RtoD (ang);
    const int      badAnglePenalty = 4 * max (0, interiorBondAngle_deg - threshhold_deg);
    return badAnglePenalty;
}

/*
+===================================================================================+
|
|
| RD_AttachPeeledBridge
|
|
|
+--------------------------------------------------------------------------------------+
| [R-] rngNo                 The ring to be merged in.
|
+ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - +
|     kAtten is a crude patch to prevent [m.n.n] systems from receiving overlapping bridges.  |
|     asUndrawnAtoms includes frozen atoms.
|
|
|
|  Penned by H.Helson, 7/12/96.
|
+==========================================================================================+
*/
void CClean::RD_AttachPeeledBridge (int rngNo)
{
      ENTER1 ("RD_AttachPeeledBridge");
      const ccCW_Sense        sense = kccCounterClockwise;
      const sdgFloat          kAtten = 0.75;
      const ccSet                    asUndrawnAtoms = RI.GetAtoms (rngNo) -
m_asRingAtomsPlaced_RDU;
      const ccSet                    asBorderAtoms = CT.Alpha_AA (asUndrawnAtoms) &
m_asRingAtomsPlaced_RDU;
      ASSERT (asBorderAtoms.NMems() == 2);
      if (asBorderAtoms.NMems() != 2) // avoid possible memory corruption by skipping out now.
            throw sdgException (sdgException::kAvoidMemCorruption, "RD_AttachPeeledBridge");
      ATOMNO  aBorder_1, aBorder_2;
      asBorderAtoms.Bits12 (aBorder_1, aBorder_2);
      ccRingTransit   hobbit (M, rngNo, cckAtom, sense);
      hobbit.MoveTo (aBorder_1);

      if (m_asRingAtomsPlaced_RDU.IsMem (hobbit.Prev()))
            Swap (aBorder_1, aBorder_2);

#ifdef _DEBUG
      hobbit.MoveTo (aBorder_1);
      ASSERT (!m_asRingAtomsPlaced_RDU.IsMem (hobbit.Prev()) &&  m_asRingAtomsPlaced_RDU.IsMem
(hobbit.Next()));
      hobbit.MoveTo (aBorder_2);
      ASSERT ( m_asRingAtomsPlaced_RDU.IsMem (hobbit.Prev()) && !m_asRingAtomsPlaced_RDU.IsMem
(hobbit.Next()));
#endif

      // Invert which side the bridge is drawn on, depending on relative lengths of the current
      // and the drawn bridge.  (CDBR-4853)
      // For example bicyclo[10.5.1]alkane: Optimally, first the seventeen-membered ring is
      // drawn perfect-polygonally, ff. by the one-membered leg.  But if a thirteen-ring is
      // first drawn, the remaining 5-leg should be drawn on the outside, not the inside.
      if (hobbit.Distance (aBorder_1, aBorder_2) - 1 < (RI.Size (rngNo) - 2) / 2)     // "-2" to
discount bridgeheads
      {
            Swap (aBorder_1, aBorder_2);
            hobbit.ReverseSense();
      }
```

```
    CDBG ( sdgOut ("Attaching peeled bridge at atoms %d (CW) and %d (CCW)\n") << aBorder_1 <<
aBorder_2; )
    const int           numAtsToDraw = asUndrawnAtoms.NMems() + 2;     // include the
border atoms
    const sdgFloat  bdLen = m_bndLen_F * kAtten;

    // If the newly placed bridge overlaps part of the ring system already laid down, try
    // increasing or decreasing its bond lengths.
    const ccSet     &asDrawnAtoms = m_asRingAtomsPlaced_RDU;
    DBG ( const char formatMsg[] = "Rating for bd len scale %3.1lf is %8.3lf (= congest[%8.3lf]
+ BdLen[%3.2lf] + bdAng[%3d (bdAng=%d)])\n"; )

    RD_BridgeCongestionEnvironment  bce_normal (*this, rngNo, hobbit, aBorder_1, aBorder_2,
asUndrawnAtoms, asDrawnAtoms, bdLen);
    const sdgFloat  congest_normal = bce_normal.ComputeCongestion();
    const int           badAnglePenalty = LinearAnglePenalty (bce_normal.m_polyPhi);    //
penalize near-linear bonds
    sdgFloat            rating_best = congest_normal + badAnglePenalty,
                          scale_best  = 1.0;
    CDBG0 ( sdgOut (formatMsg) << scale_best << rating_best << congest_normal << 0. <<
badAnglePenalty << 180-RtoD (bce_normal.m_polyPhi); )
    if (bce_normal.IsCongested())
    {
        for (sdgFloat scale = 0.5;  scale < 2.0;  scale += 0.2) // misses scale=1.0, which
was covered above
        {
            RD_BridgeCongestionEnvironment  bce_trial (*this, rngNo, hobbit, aBorder_1,
aBorder_2, asUndrawnAtoms, asDrawnAtoms, bdLen * scale);
            const sdgFloat  congest_trial = bce_trial.ComputeCongestion();
```

```
                    const sdgFloat  nonStandardBondLengthPenalty = 80 * abs (scale - 1.0);  //
arbitrary penalty for not using std bond length
                    const int              badAnglePenalty = LinearAnglePenalty
(bce_trial.m_polyPhi);
                    const sdgFloat  rating_trial = congest_trial + nonStandardBondLengthPenalty
+ badAnglePenalty;
                    CDBG0 ( sdgOut (formatMsg) << scale << rating_trial << congest_trial <<
nonStandardBondLengthPenalty
                                        << badAnglePenalty << 180-RtoD
(bce_trial.m_polyPhi); )
                    if (rating_trial < rating_best)
                    {
                          rating_best = rating_trial;
                          scale_best  = scale;
                    }
              }
       }
       CDBG ( sdgOut ("Ring%2d: Best bridge scale factor = %3.2lf (rating = %8.3lf)\n") << rngNo
<< scale_best << rating_best; )

       RD_AttachThing (rngNo, aBorder_1, aBorder_2, hobbit, aBorder_1, aBorder_2, numAtsToDraw,
bdLen * scale_best);

}



/*
```

```
     File:        :sdg:sdg_repo.cpp

     Contains:     Repositions fragments after they are designed de novo.

     Copyright:     © 1998-2000 CambridgeSoft Corp., all rights reserved.

     $Header: /ChemDraw/Src/sdg/sdg_repo.cpp 28    12/23/99 6:32p Jsb $

+--------------------------------------------------------------------------------------+
HEH  07/12/99  [Pending] Reposition_Analytic(): Reposition all fragments, not just redrawable
ones.  |
HEH  05/09/99  Reposition(): now available on request by the kReposition opflag.
|
HEH  03/15/99  Reposition_Analytic(): Fix mem err caused by incorrect dimensioning.         |
HEH  01/22/99  Reposition_Analytic(): Avoid divide-by-zero when m_stdBondLen_W provided zero.|
SDR  01/05/99  Avoid conflict with Mac toolbox "topLeft" macro
|
HEH  12/05/98  Reposition_Analytic(): When no bonds, use standard bond length.
|
HEH  11/30/98  CDBR-3905: Added Reposition_Analytic() using simple "dynamic grid" alg.       |
HEH  11/30/98  CAMEO's FreeRect algorithm adapted for use here (in C++) as sdgFreeRect::     |
HEH  11/30/98  File created.
|
+--------------------------------------------------------------------------------------+
*/

#include "sdg.h"
#include <limits>
using namespace std;
```

```
SET_OUTPUT_LEVEL2 (0, &SDG::sdgMasterOutputLevel)

/*
+===================================================================================+
| sdgFreeRect   This class is used to locate a fre rectangle in a given target area and a     |
|                    list of rectangles that are off limits.  Represents a given
rectangulaar   |
|                    area (A) as a patchwork of "free" (available) rectangles.  These
rectangles     |
|                    do overlap and all reside within A.
|
+===================================================================================+
*/
class sdgFreeRect       // Method adapted from this author's approach in CAMEO's DynaJump routine.
{
public:
     // METHODS
                              sdgFreeRect (ccRect targetRectangle = ccRect (0,0,0,0),
bool mergeAdjacentRects = true);
     bool                RegisterOccupiedRectangle (const ccRect &occRect);

     typedef list<ccRect>::iterator  FrIter;

private:
     void                AddFR (FrIter pInsertBefore, long left, long top, long right, long
bottom);
     void                DelFR (FrIter pFR);
DBG(void                    Dump (FrIter pFR); )
DBG(void              Dump(); )
DBG(void*              FrAddress (const FrIter pIt); )

    // DATA
public:
    list<ccRect>   m_freeRects;

private:
    ccRect             m_targetRect;
    unsigned long   m_numPasses;    // for debugging only
    bool               m_merge;
    static bool          m_skipSmall;
};
bool   sdgFreeRect::m_skipSmall = true;// don't waste time with very small regions


/*
+==========================================================================================+
| sdgFreeRect ctor
|
+==========================================================================================+
*/
sdgFreeRect::sdgFreeRect (ccRect targetRectangle, bool mergeAdjacentRects)
    :     m_targetRect    (targetRectangle)
    ,     m_merge              (mergeAdjacentRects)
    ,     m_numPasses          (0)
{
    if (m_targetRect.IsRectEmpty())
    {
        const long      kMin = -numeric_limits<long>::min() / 4,
                         kMax =  numeric_limits<long>::max() / 4;
        m_targetRect.Set (kMin, kMin, kMax, kMax);
        m_merge = false;
    }
    AddFR (m_freeRects.end(), m_targetRect.left, m_targetRect.top, m_targetRect.right,
m_targetRect.bottom);
}

/*
+==========================================================================+
| AddFR       Create a new FR.  The new FR is inserted just before pInsertBefore.
|
+==========================================================================+
*/
void sdgFreeRect::AddFR (FrIter pInsertBefore, long left, long top, long right, long bottom)
{
    ENTER1 ("AddFR");
    const double   kPercentage = .05;            // percentage of dimension.

    // Check if new FR is entirely within an already existing one.
    // This should not occur if the algorithm is functioning properly.
    for (FrIter check = m_freeRects.begin();  check != m_freeRects.end();  check++)
    {
        if (check == pInsertBefore)     // pInsertBefore IS a superset; however it may
perish shortly
            continue;
        bool   inside = false;
        if (check->left <= left  &&  check->right  >= right)
```

```
            if (check->top <= top && check->bottom >= bottom)
                  inside = true;
      if (check->left >= left   &&  check->right  <= right)
            if (check->top >= top &&  check->bottom <= bottom)
                  inside = true;
      if (inside)
      {
            CDBG1 ( sdgOut ("Proposed FR (%3ld..%3ld)(%3ld..%3ld) is inside preexisting
FR %lX; pInsertBefore=%lX\n") <<
                                    left << right << top << bottom << FrAddress
(check) << FrAddress (pInsertBefore); )
            return;
      }

      // Check if new FR borders a preexisting one; if so, merge.
      if (!m_merge)
            continue;
      const long     width  =  right - left;
      const long     height = bottom - top;
      if (abs (check->left  - left)  < (int)(kPercentage * (float)width)  &&
            abs (check->right - right) < (int)(kPercentage * (float)width))
      // Horizontal dimension aligns; check for overlap in V
      {
            if (check->bottom >= bottom)
            {
                  if (check->top <= bottom)      // overlap in Y?: new FR is above &
overlapping
                  {
#ifdef _DEBUG
```

```
                    CDBG2 ( sdgOut << "Merging two FR's #1" << endl; )
                    CDBG2 ( sdgOut ("Merging new (%ld..%ld, %ld..%ld) with old
%lX; Before:\n") << left << right << top << bottom << FrAddress (check); )
                    CDBG2 ( Dump (check); )
#endif
                    check->left  = max (check->left,  left);        // use
smaller rect in H
                    check->right = min (check->right, right);
                    check->top   = min (check->top,   top);          // possibly
expand upwards
                    CDBG2 ( sdgOut << "After:\n"; Dump (check); )
                    return; // bottom stays the same
                }
            }
            else if (top <= check->bottom)  // new FR is below & overlapping "check"
            {
                check->left        = max (check->left,        left);
                check->right    = min (check->right,    right);
                check->bottom   = max (check->bottom,   bottom);        // possibly
expand downwards
                CDBG2 ( sdgOut << "Merging two FR's #2" << endl; )
                return;        // top stays the same
            }
        }

        if (abs (check->top    - top)    < kPercentage * height &&
            abs (check->bottom - bottom) < kPercentage * height)
        // Vertical dimension aligns; check for overlap in H
        {
                if (check->right >= right)
                {
                        if (check->left <= right)// overlap in X? (new is to left of check,
but overlapping)
                        {
                                check->top = max (check->top, top);     // use smaller rect
in V
                                check->bottom = min (check->bottom, bottom);
                                check->left = min (check->left, left);  // expand leftwards
                                CDBG2 ( sdgOut << "Merging two FR's #3" << endl; )
                                return; // right stays the same
                        }
                }
                else if (left <= check->right)  // new FR is to right of "check" but
overlapping
                {
                        check->top    = max (check->top,    top);
                        check->bottom = min (check->bottom, bottom);
                        check->right  = max (check->right, right);      // expand rigtwards
                        CDBG2 ( sdgOut << "Merging two FR's #4" << endl; )
                        return;         // left stays the same
                }
        }
    }

    if (m_skipSmall)
    {
        if ((right-left) < 20  ||  (bottom-top) < 20)
            return;
        if ((double)(right-left) * (double)(bottom-top) < 900)
            return; // an icon is 32x32
    }

    // Insert the new FR.
    DBG ( FrIter    pNew = )
    m_freeRects.insert (pInsertBefore, ccRect (left, top, right, bottom));

    CDBG0 ( sdgOut ("Created %X: (%ld..%ld, %ld..%ld)\n") << FrAddress (pNew) << left << right
<<top << bottom; )
} // AddFR()


/*
+===============================================================================+
| DelFR
|
+===============================================================================+
*/
void sdgFreeRect::DelFR (FrIter pFR)
{
    CDBG1 ( sdgOut ("DelFR: %X\n") << FrAddress (pFR); )
    m_freeRects.erase (pFR);
    CDBG2 ( sdgOut << "New list is:\n";  Dump(); )
}


/*
```

```
+===================================================================================+
| RegisterOccupiedRectangle    "Apply" a screen object (represented by rectangle occupRect)   |
|                                  to the registered Free Rectangles.  Return
False iff infinite     |
|                                  loop detected (merely a precaution).
|
+===================================================================================+
*/
bool sdgFreeRect::RegisterOccupiedRectangle (const ccRect &occupRect)
{
    ENTER1 ("RegisterOccupiedRectangle");
    CDBG1 ( sdgOut ("(%3ld..%3ld, %3ld..%3ld)\n") << occupRect.left << occupRect.right <<
occupRect.top << occupRect.bottom; )
    DBG ( m_numPasses++; )

    for (FrIter pCur = m_freeRects.begin();  pCur != m_freeRects.end();  )
    {
        FrIter  pNext = pCur;  pNext++; // squirrel value since pCur may get destroyed

        CDBG0 ( sdgOut ("Comparing occupied rect with FR %lX (%3ld..%3ld,%3ld..%3ld)\n") <<
                    FrAddress (pCur) << pCur->left << pCur->right << pCur->top <<
pCur->bottom; )

        if (occupRect.right > pCur->left)
        {
            if (occupRect.left < pCur->right)      // overlap in X
            {
                if (occupRect.bottom > pCur->top)
                {
```

```
                                if (occupRect.top < pCur->bottom)       // overlap in Y:
bingo!
                                {
                                        if (pCur->left < occupRect.left)
                                                AddFR (pCur, pCur->left, pCur->top,
occupRect.left, pCur->bottom);
                                        if (pCur->right > occupRect.right)
                                                AddFR (pCur, occupRect.right, pCur->top,
pCur->right, pCur->bottom);
                                        if (pCur->top < occupRect.top)
                                                AddFR (pCur, pCur->left, pCur->top,
pCur->right, occupRect.top);
                                        if (pCur->bottom > occupRect.bottom)
                                                AddFR (pCur, pCur->left, occupRect.bottom,
pCur->right, pCur->bottom);
                                        DelFR (pCur);
                                }
                        }
                }
        }
        pCur = pNext;
    } // pCur
    return true;
} // RegisterOccupiedRectangle()


//-----------------------------------------------------------------------------------------
//          Utility functions used by Reposition_Analytic()
```

```
//------------------------------------------------------------------------------------------
inline long DistFromCenter_1_Dimension (long edge_1, long edge_2, long center = 0)
{
	ASSERT (edge_2 >= edge_1);
#if 0
	if (Within (center, edge_1, edge_2))
		return 0;
	return min (abs (edge_1 - center), abs (edge_2 - center));
#else
	if (edge_2 < center)
		return center - edge_2;
	if (edge_1 > center)
		return edge_1 - center;
	return 0;
#endif
}
//------------------------------------------------------------------------------------------
inline long DistFromCenter (const ccRect &rect, long center_x = 0, long center_y = 0)
{
	long	dx = DistFromCenter_1_Dimension (rect.left, rect.right,  center_x),
			dy = DistFromCenter_1_Dimension (rect.top , rect.bottom, center_y);
	return dx * dx + dy * dy;
}
//------------------------------------------------------------------------------------------
static ccRect   ScaleAndCenter (double width, double height, double scalingFactor)
{
	width  *= scalingFactor;
	height *= scalingFactor;
	ccRect  result;
```

```
    result.left   = -Round (width / 2);
    result.right  =  Round (width / 2);
    result.top    = -Round (height / 2);
    result.bottom =  Round (height / 2);
    return result;
}
//------------------------------------------------------------------------------------------


/*
+===========================================================================================+
|
|
| Reposition    Place fragments on-screen and spaced apart after they are redrawn.
|
|
|
+ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - +
|    Analytic repositioning is only applied if drawing de novo, since it destroys the
|
|    relative positions of molecules.  Dynamic repositioning is performed regardless.
|
+===========================================================================================+
*/
void SDG_Whole_PostProcessing::Reposition()
{
    if (PD.GetNFrags() <= 1)
        return;
    // Analytic Repositioning
    if (OpFlagged (kIgnoreCoordinates)  ||  OpFlagged (kReposition))
        Reposition_Analytic();

    // Dynamic Repositioning
    Reposition_Dynamic();

} // Reposition



/*
+==================================================================================+
|
|
| Reposition_Analytic   The analytic repositioning procedure.
|
|
|
+ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - +
|    ALGORITHM
|
|
|
|    1.    Rank fragments by decreasing size.
|
|    2.    In order of decreasing size:
|
|          a.     Find free rectangle that is closest to center (0,0) and large enough to
|
|                 accommodate the fragment.
|
|          b.     Place the fragment there, as close as possible to the center.
|
|          c.     Recenter the fragments so they center on the origin (0,0).  Or
equivalently,  |
|                 track the new central position, defined as the center of the smallest
bounding       |
|                 rectangle of the placed fragments.
|
|
|
+==========================================================================================+
*/
void SDG_Whole_PostProcessing::Reposition_Analytic()
{
/*
+ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - +
|     Remap the molecular coordinate system to a nice, large integral coordinate space that   |
|     the Free Rectangle class can use.  The present molecules' scaling may be anything,
|
|     from very tiny exponentials to very large ones.  Moreover, different fragments might   |
|     in principle reside wildly far apart, or superimposed.  We do expect, however, that
|
|     they are all scaled similarly.  That is, if one molecule's bond lengths are about
|
|     7*10-3, then the other molecules' bonds fall in the same ballpark.
```

```
|
+----------------------------------------+
*/
	ENTER0 ("Reposition_Analytic");
	const double	avgBndLen = (NB == 0) ? (m_stdBondLen_W < 1.0E-12 ? 100. : m_stdBondLen_W)
: M.MedianBondLength (NULL, kIn2D);
	const long		kIntegralBondLength = 10;	// A nice, well-behaved integral
bond length
	const double	scalingFactor = kIntegralBondLength / avgBndLen;
	DBG ( if (0) )
	CDBG0 (  { sdgOut << "Before AnaRepo:\n"; DumpCoords (kAllFragments); sdgOut
("scalingFactor = %6.2lf\n") << scalingFactor; } )
	ccSet	sFragsToPlace (PD.GetNFrags());
	sFragsToPlace.Fill();
sFragsToPlace = m_FrgsToPlace;  // this line makes it compatible w/old behavior; remove this line
at some point -heh 7/27/99.

	// Rank by decreasing size
	multimap<long,int>	bySize;
	int			frgNum;
	vector<ccRect>	frgDimensions (sFragsToPlace.Last() + 1);	// Integral fragment
position
	ccVec2D			maxTargetSpace (kIntegralBondLength, kIntegralBondLength);	//
Start off with some small number
	LOOP_SET (sFragsToPlace, frgNum)
	{
		double		dx, dy;
		M.GetSize (&dx, &dy, &PD.GetFragAtms (frgNum));
		ccRect		minmax = ScaleAndCenter (dx, dy, scalingFactor);
```

```
            minmax.InflateRect (kIntegralBondLength / 2, kIntegralBondLength / 2);  // add a
half-bond length margin all about each molecule
            const long      area = 4 * minmax.right * minmax.bottom;
            bySize.insert (pair<const long,int> (-area, frgNum));   // Use negative area to get
automatic sorting by decreasing size.
            frgDimensions [frgNum] = minmax;

            maxTargetSpace.x += dx * scalingFactor + kIntegralBondLength;
            maxTargetSpace.y += dy * scalingFactor + kIntegralBondLength;
      }

      const ccRect    targetSpace (-maxTargetSpace.x, -maxTargetSpace.y, maxTargetSpace.x,
maxTargetSpace.y);
      sdgFreeRect     FRs (targetSpace, false);
      ccRect  usedRect (0,0,0,0);     // Describes the limits of placed fragments.
      long    center_x = (targetSpace.left + targetSpace.right ) / 2, // Shorthand for the center
of usedRect.  Will shift as we feed fragments.
                  center_y = (targetSpace.top  + targetSpace.bottom) / 2;

      for (multimap<long,int>::iterator  it = bySize.begin();  it != bySize.end();   )
      {
            const int       frgNum = it->second;
            ccRect          &curFragRect = frgDimensions [frgNum];
            const long       width  = curFragRect.Width(),
                              height = curFragRect.Height();

            CDBG1 ( sdgOut ("Placing fragment %ld (area %ld): width = %ld; height = %ld\n") <<
                                    frgNum << -it->first << width << height; )
```

```
        // Find free rectangle large enough and closest to center
        ccRect  best;
        bool    foundBest = false;
        for (sdgFreeRect::FrIter pFR = FRs.m_freeRects.begin();  pFR !=
FRs.m_freeRects.end();  pFR++)
        {
            if (pFR->Width() < width  ||  pFR->Height() < height)
                continue;
            if (!foundBest  ||  DistFromCenter (*pFR, center_x, center_y) <
DistFromCenter (best, center_x, center_y))
            {
                foundBest = true;
                best = *pFR;
            }
        }
        if (!foundBest)
        {
            ASSERT (false); // shouldn't happen
            best = *FRs.m_freeRects.begin();
        }
        CDBG1 ( sdgOut ("Free spot to place fragment is (%ld..%ld, %ld..%ld)\n") <<
best.left << best.right << best.top << best.bottom; )

        /*
        +-----------------------------------------
+
        |   Translate the current fragment within the "best" rectangle so that it is as
|
        |   close to center as possible.  This involves locating the two sides closest
```

```
to    |
          |    the center.  For either dimension (vertical or horizontal), there are three
|
          |    cases:
|
          |         a.    "Best"'s Low value is closest to center.  Set fragment's
low value to   |
          |               this, and its hight value to Low + width/height.
|
          |         b.    "Best"'s High value is closest to center.  Set fragment's
high value to  |
          |               this, and its low value to High - width/height.
|
          |         c.    "Best"'s Low and High values flank the center.  Center
fragment's low  |
          |               and high values on the center.
|
          +----------------------------------------
+
          */
          if (best.left > center_x - width / 2)
          {
               curFragRect.left  = best.left;
               curFragRect.right = best.left + width;
          }
          else if (best.right < center_x + width / 2)
          {
               curFragRect.right = best.right;
               curFragRect.left  = best.right - width;
```

```
        }
        else
        {
              curFragRect.left  = center_x - width/2;
              curFragRect.right = center_x + width/2;
        }

        if (best.top > center_y - height / 2)
        {
              curFragRect.top    = best.top;
              curFragRect.bottom = best.top + height;
        }
        else if (best.bottom < center_y + height / 2)
        {
              curFragRect.bottom = best.bottom;
              curFragRect.top    = best.bottom - height;
        }
        else
        {
              curFragRect.top    = center_y - height/2;
              curFragRect.bottom = center_y + height/2;
        }
        CDBG1 ( sdgOut ("Fragment slid to position (%ld..%ld, %ld..%ld)\n") <<
curFragRect.left
                    << curFragRect.right << curFragRect.top << curFragRect.bottom; )
        ASSERT (Within (curFragRect.left, best.left, best.right)   &&  Within
(curFragRect.right,  best.left, best.right));
        ASSERT (Within (curFragRect.top,  best.top,  best.bottom)  &&  Within
(curFragRect.bottom, best.top,  best.bottom));
```

```
        it++;

        // Insinuate the used rectangle on the free rectangle model.
        if (it != bySize.end()) // no point if this is the last fragment
              FRs.RegisterOccupiedRectangle (curFragRect);

        // Update the limits of the placement rectangle, and its center.
        usedRect |= curFragRect;
        center_x = (usedRect.left + usedRect.right)  / 2;
        center_y = (usedRect.top  + usedRect.bottom) / 2;
        CDBG1 ( sdgOut ("center is now (%d,%d)\n") << center_x << center_y; )
    }

#ifdef _DEBUG
    int     i,j;   // Ensure we succeeded in spacing the frags apart
    LOOP_SET (sFragsToPlace, i)
          LOOP_SET2 (sFragsToPlace, i, j)
                ASSERT (!frgDimensions [i].Intersects (frgDimensions [j]));
#endif

    // Translate the real fragments, preserving the current center.
    ccVec2D SER_min, SER_max;
    ccSet   asAllAtoms (NA); asAllAtoms.Fill();
    SmallestBoundingRect (SER_min, SER_max, asAllAtoms);
    ccPoint2D      cent ((SER_min.x + SER_max.x) / 2,
                         (SER_min.y + SER_max.y) / 2);
    ccVec2D center_offset (cent.x - center_x / scalingFactor,
                           cent.y - center_y / scalingFactor);
    CDBG1 ( sdgOut ("\nBeginning translation.  center_offset_x/_y = (%lf,%lf)\n") <<
center_offset.x << center_offset.y; )
      LOOP_SET (sFragsToPlace, frgNum)
      {
            ccVec2D              oldFrag_min, oldFrag_max;
            SmallestBoundingRect (oldFrag_min, oldFrag_max, PD.GetFragAtms (frgNum));
            const ccRect    &curFragRect = frgDimensions [frgNum];
            const ccPoint2D topLeftPt ((curFragRect.left + kIntegralBondLength/2) /
scalingFactor,
                                                  (curFragRect.top  +
kIntegralBondLength/2) / scalingFactor);
            const ccVec2D   dxy = topLeftPt - oldFrag_min;
            // topLeftPt is the point to which we wish to translate the top left corner.
            ccTranslate (M, dxy.x + center_offset.x, dxy.y + center_offset.y, 0.,
&PD.GetFragAtms (frgNum));
            CDBG2 ( sdgOut ("After translating fragment %d:\n") << frgNum;  DumpCoords
(kAllFragments); )
      }
      if (s_dbgFlags.GR_tracing)      { sdgOut << "Reposition_Analytic: Ending molecule is:\n";
DumpCoords (kAllFragments); }

}
```